

## LETTER

# A Concurrent Model for Imperative Languages with Improved Atomicity

Keehang KWON<sup>†</sup>, *Member* and Daeseong KANG<sup>††</sup>, *Nonmember*

**SUMMARY** We propose a new concurrent model for imperative languages where concurrency occurs at a subprogram level. This model introduces a new *block sequential* statement of the form  $\sharp(G_1, \dots, G_n)$  where each  $G_i$  is a statement. This statement tells the machine to execute  $G_1, \dots, G_n$  sequentially and atomically (*i.e.*, without interleaving). It therefore enhances atomicity and predictability in concurrent programming.

We illustrate our idea via  $C^\parallel$ , an extension of the core concurrent C with the new block sequential statement.

**key words:** concurrency, imperative languages, block sequential.

## 1. Introduction

Adding concurrency to imperative programming – C, its extension [6] and Java – is an attractive task. While concurrent programming may appear to be a simple task, it has proven too difficult to use, predict or debug.

An analysis shows that this difficulty comes from the fact that interleavings among threads – which are typically done by OS schedulers – are quite arbitrary and cannot be controlled at all by the programmer.

We observe that concurrent programming can be made easier by making interleaving less arbitrary and more controlled. Inspired by [4], [5], we propose two ideas for this. First, we propose a concurrent model with its own embedded scheduler. Our scheduler works on a high-level: it allows the execution to switch from one assignment statement to another. This is in contrast to OS schedulers which allows the execution to switch from one machine instruction to another. For example, the assignment statement  $c = c + 1$  is *atomic* in the sense that it runs to completion without being interleaved. This reduces nondeterminism and error known as *transient* errors.

Second, semaphores and monitors are typically used as facilities for mutual exclusion. We propose a simpler method to solve mutual exclusion. Toward this end, we propose a new *block sequential* statement  $\sharp(G_1, \dots, G_n)$ , where each  $G_i$  is a statement. This has the following execution semantics: execute  $G_1, \dots, G_n$  sequentially and consecutively (*i.e.*, without being interleaved). In other words, our interpreter

treats  $\sharp(G_1, \dots, G_n)$  atomic. This can be easily implemented by designing our interpreter working in two different modes: the *concurrent* mode and the traditional *sequential* mode. Thus, if our interpreter encounters  $\sharp(G_1, \dots, G_n)$ , it switches from the concurrent mode to the sequential mode and proceeds just like the traditional C interpreter. In this way, atomicity of this statement is guaranteed. We intend to use this construct to each critical region.

This paper focuses on the minimum core of C, enhanced with concurrency at a subprogram level. This is to present the idea as concisely as possible. The remainder of this paper is structured as follows. We describe  $C^\parallel$ , an extension of concurrent C with a new statement in Section 2. In Section 3, we present an example of  $C^\parallel$ . Section 4 concludes the paper.

## 2. The Language

The language is core C with procedure definitions. It is described by  $G$ - and  $D$ -formulas given by the syntax rules below:

$$\begin{aligned} G &::= \text{true} \mid A \mid x = E \mid ;(G_1, \dots, G_n) \mid \\ &\quad \sharp(G_1, \dots, G_n) \\ D &::= A = G \mid \forall x D \end{aligned}$$

In the above,  $A$  represents a head of an atomic procedure definition of the form  $p(x_1, \dots, x_n)$  where  $x_1, \dots, x_n$  are parameters. A  $D$ -formula is called a procedure definition. In the transition system to be considered, a  $G$ -formula will function as a thread and a set of  $G$ -formulas (*i.e.*, a set of threads) will function as the main statement, and a set of  $D$ -formulas enhanced with the machine state (a set of variable-value bindings) will constitute a program. Thus, a program is a union of two disjoint sets, *i.e.*,  $\{D_1, \dots, D_n\} \cup \theta$  where each  $D_i$  is a  $D$ -formula and  $\theta$  represents the machine state. Note that  $\theta$  is initially set to an empty set and will be updated dynamically during execution via the assignment statements.

We will present an interpreter for our language via a proof theory [3], [7]–[9]. This is in contrast to other complex approaches to describing an interpreter for concurrent languages [1], [2]. Note that our interpreter alternates between the concurrent execution phase and the backchaining phase. In the concurrent execution

Manuscript received January 1, 2003.

Manuscript revised January 1, 2003.

Final manuscript received January 1, 2003.

<sup>†</sup>The authors are with Computer Eng., DongA University. email:khkwon@dau.ac.kr

<sup>††</sup>The author is with Electronics Eng., DongA University.

phase (denoted by  $ex(\mathcal{P}, \|( \Gamma, G, \Delta), \mathcal{P}' )$ ) it tries to select and execute a thread  $G$  among a set of threads  $(\Gamma, G, \Delta)$  with respect to a program  $\mathcal{P}$  and produce a new program  $\mathcal{P}'$  by reducing  $G$  to simpler forms until  $G$  becomes an assignment statement, true or a procedure call. The rules (4)-(11) deal with this phase. Here both  $\Gamma$  and  $\Delta$  denote a set of  $G$ -formulas. If  $G$  becomes a procedure call, the interpreter switches to the backchaining mode. This is encoded in the rule (3). In the backchaining mode (denoted by  $bc(D, \mathcal{P}, A, \mathcal{P}', \Gamma, \Delta)$ ), the interpreter tries to solve a procedure call  $A$  and produce a new program  $\mathcal{P}'$  by first reducing a procedure definition  $D$  in a program  $\mathcal{P}$  to its instance (via rule (2)) and then backchaining on the resulting definition (via rule (1)). To be specific, the rule (2) basically deals with argument passing: it eliminates the universal quantifier  $x$  in  $\forall x D$  by picking a value  $t$  for  $x$  so that the resulting instantiation, written as  $[t/x]D$ , matches the procedure call  $A$ . The notation  $S$  seqand  $R$  denotes the sequential execution of two tasks. To be precise, it denotes the following: execute  $S$  and execute  $R$  sequentially. It is considered a success if both executions succeed. Similarly, the notation  $S$  parand  $R$  denotes the parallel execution of two tasks. To be precise, it denotes the following: execute  $S$  and execute  $R$  in any order. Thus, the execution order is not important here. It is considered a success if both executions succeed. The notation  $S \leftarrow R$  denotes reverse implication, i.e.,  $R \rightarrow S$ .

**Definition 1.** Let  $\|(G_1, \dots, G_n)$  be a sequence of threads to run concurrently and let  $\mathcal{P}$  be a program. Then the notion of executing  $\langle \mathcal{P}, \|(G_1, \dots, G_n) \rangle$  concurrently and producing a new program  $\mathcal{P}'$  –  $ex(\mathcal{P}, \|(G_1, \dots, G_n), \mathcal{P}')$  – is defined as follows:

- (1)  $bc((A = G_1), \mathcal{P}, A, \mathcal{P}_1, \Gamma, \Delta) \leftarrow ex(\mathcal{P}, \|( \Gamma, G_1, \Delta), \mathcal{P}_1)$ . % A matching procedure for  $A$  is found.
- (2)  $bc(\forall x D, \mathcal{P}, A, \mathcal{P}_1, \Gamma, \Delta) \leftarrow bc([t/x]D, \mathcal{P}, A, \mathcal{P}_1, \Gamma, \Delta)$ . % argument passing
- (3)  $ex(\mathcal{P}, \|( \Gamma, A, \Delta), \mathcal{P}_1) \leftarrow (D \in \mathcal{P} \text{ parand } bc(D, \mathcal{P}, A, \mathcal{P}_1, \Gamma, \Delta))$ . %  $A$  is a procedure call
- (4)  $ex(\mathcal{P}, \|(true), \mathcal{P})$ . % True is always a success.
- (5)  $ex(\mathcal{P}, \|( ), \mathcal{P})$ . % Empty threads mean a success.
- (6)  $ex(\mathcal{P}, \|( \Gamma, x = E, \Delta), \mathcal{P}_1) \leftarrow eval(\mathcal{P}, E, E') \text{ seqand } \% \text{ evaluate } E \text{ to get } E' (ex(\mathcal{P} \uplus \{ \langle x, E' \rangle \}, \|( \Gamma, \Delta), \mathcal{P}_1))$   
 % If an assignment statement  $x = E$  is chosen by our interpreter, update  $x$  to  $E'$  and return to the interpreter. Here,  $\uplus$  denotes a set union but  $\langle x, V \rangle$  in  $\mathcal{P}$  will be replaced by  $\langle x, E' \rangle$ .
- (7)  $ex(\mathcal{P}, \|( \Gamma, ; ( ), \Delta), \mathcal{P}_1) \leftarrow$

$ex(\mathcal{P}, \|( \Gamma, \Delta), \mathcal{P}_1))$ . % an empty sequential composition is a success.

- (8)  $ex(\mathcal{P}, \|( \Gamma, ; (G_1, \dots, G_m), \Delta), \mathcal{P}_2) \leftarrow (ex(\mathcal{P}, \|(G_1), \mathcal{P}_1) \text{ seqand } ex(\mathcal{P}_1, \|( \Gamma, ; (G_2, \dots, G_m), \Delta), \mathcal{P}_2))$ . %  
 % If a sequential composition  $; (G_1, \dots, G_m)$  is chosen, execute  $G_1$  in sequential mode and then return to the interpreter with the rest.
- (9)  $ex(\mathcal{P}, \|( \Gamma, repeat(G), \Delta), \mathcal{P}_2) \leftarrow (ex(\mathcal{P}, \|(G), \mathcal{P}_1) \text{ seqand } ex(\mathcal{P}_1, \|( \Gamma, repeat(G), \Delta), \mathcal{P}_2))$ .  
 % If a repeat statement  $repeat(G)$  is chosen, execute  $G$  in sequential mode and then return to the interpreter with the rest plus  $repeat(G)$ .
- (10)  $ex(\mathcal{P}, \|( \#( ), \mathcal{P})$ .  
 % An empty block sequential statement is a success.
- (11)  $ex(\mathcal{P}, \|( \Gamma, \#(G_1, \dots, G_m), \Delta), \mathcal{P}_3) \leftarrow (ex(\mathcal{P}, \|(G_1), \mathcal{P}_1) \text{ seqand } (ex(\mathcal{P}_1, \|( \#(G_2, \dots, G_m), \Delta), \mathcal{P}_2) \text{ seqand } ex(\mathcal{P}_2, \|( \Gamma, \Delta), \mathcal{P}_3))$ .  
 % If a block sequential statement  $\#(G_1, \dots, G_m)$  is chosen, execute  $G_1$  in sequential mode and then  $\#(G_2, \dots, G_m)$  in sequential mode and then return to the interpreter (which proceeds in concurrent mode) with the remaining.

If  $ex(\mathcal{P}, G, \mathcal{P}_1)$  has no derivation, then the interpreter returns the failure. Initially it works in the concurrent mode. In the concurrent mode, we assume that our interpreter chooses a thread using some predetermined algorithm. Note that executing *one* thread  $G$  concurrently, denoted by  $\|(G)$ , is identical to executing  $G$  sequentially.

### 3. Examples

As a well-known example, we examine the system that allows people to sign up for a mailing list. An example of this class is provided by the following code where the procedure below adds a person to a list:

% Procedure signup

signup(person) =  
 ( $N = N + 1 \# list[N] = person$ ) % critical section

and the main program consists of two concurrent threads as follows:

$\|(signup(tom), signup(bill))$

In the above, we used a more traditional notation  $(G_1\sharp, \dots, \sharp G_n)$  instead of  $\sharp(G_1, \dots, G_n)$ .

Although the above signup procedure is in fact a critical section, it works correctly because no interleaving – due to the presence of  $\sharp$  – is allowed in the critical section. Note that our code is very concise compared to the traditional ones using semaphores.

#### 4. Conclusion

In this paper, we proposed a simple concurrent model for imperative languages. This model introduces a block sequential statement  $\sharp(G_1, \dots, G_n)$  where each  $G_i$  is a statement. This statement executes  $G_1, \dots, G_n$  sequentially and atomically. It therefore enhances atomicity and predictability.

Although we focused on a simple concurrent model for imperative language at a subprogram level, it seems possible to apply our ideas to existing concurrent and parallel computing models [1], [2].

#### 5. Acknowledgements

This work was supported by Dong-A University Research Fund.

#### References

- [1] J. Alglave and L. Maranget and M. Tautschnig, “Herding cats: Modelling, simulation, testing and data mining for weak memory”, *ACM Transactions on Programming Languages and Systems*, vol.36, no.2, pp.1–74, 2014.
- [2] G. Boudol and G. Petri, “Relaxed memory models: an operational approach”, In *POPL*, pp.392–403, ACM, 2009.
- [3] G. Kahn, “Natural Semantics”, In the 4th Annual Symposium on Theoretical Aspects of Computer Science, LNCS vol. 247, 1987.
- [4] G. Japaridze, “Introduction to computability logic”, *Annals of Pure and Applied Logic*, vol.123, pp.1–99, 2003.
- [5] G. Japaridze, “Sequential operators in computability logic”, *Information and Computation*, vol.206, No.12, pp.1443–1475, 2008.
- [6] K. Kwon, S. Hur and M. Park, “Improving Robustness via Disjunctive Statements in Imperative Programming”, *IEICE Transactions on Information and Systems*, vol.E96-D, No.9, pp.2036–2038, September, 2013.
- [7] J. Hodas and D. Miller, “Logic Programming in a Fragment of Intuitionistic Linear Logic”, *Information and Computation*, vol.110, No.2, pp.327–365, 1994.
- [8] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov, “Uniform proofs as a foundation for logic programming”, *Annals of Pure and Applied Logic*, vol.51, pp.125–157, 1991.
- [9] D. Miller, G. Nadathur, *Programming with higher-order logic*, Cambridge University Press, 2012.